Is Eight Enough ? - The Eight Queens Problem Re-examined

John S. Gray
Department of Math/Physics/Computer Science
University of Hartford
West Hartford, CT 06117
email: gray@uhasun.hartford.edu

Abstract

A detailed analysis of a classic backtracking problem, The Eight Queen Problem is presented. The paper addresses additional facets of the Eight Queen Problem that might be overlooked when casually generating a program solution. The author suggests that the extra time taken to fully analyze the problem will result in a better understanding of the problem which in turn will manifest itself in a better program solution.

1. Introduction

During the course of a semester a computer science instructor will assign a number of programming problems to students. While problem sets for introductory courses focus on syntax and basic programming constructs, advanced courses tend to address more complex problems. The problems assigned to students in advanced classes can be grouped into two broad categories. Those problems that are minor variations of problems that are discussed in the text (for which the author has often provided a program solution) and those problems that can be termed as **interesting,** related problems (members of the same problem class) for which no complete solution discussion or author generated program is readily available. In an attempt to challenge students and foster independent, creative thinking, instructors have the tendency to lean toward assigning problems that fall into the latter category not the former. Unfortunately, time often precludes the instructor from generating his own detailed problem analysis / evaluations and program solutions to such problems. The student, if not specifically directed to do so, will more times than not, fail to research the problem before embarking on his own solution. In their press to meet assignment deadlines, both instructors and students alike may miss the insights that can result from a detailed, deliberate analysis of the problem.

The problem discussion that follows addresses a programming problem assignment that falls into the **interesting** category. While some text authors provide a basic discussion of the problem (and on rare occasions may offer a program solution), most authors often do not have the time or space to devote to an in-depth analysis of the problem. It is the hope of the author that the materials presented below will serve both as a springboard for new ideas and approaches to an old problem for future computer science students and as a reference for computer science instructors.

2. Problem Description

The problem, initially investigated by C. F. Gauss in 1850 and often briefly described in college level data structures texts in the section on backtracking (N. Wirth [3], R. Kruse [1]), seems straight forward. Place eight queens on an 8 x 8 chess board so that no queen is in check by any other queen. For non-chess aficionados a queen may move an arbitrary number of squares horizontally, vertically or on the diagonal.

3. Problem Analysis

There is nothing special about using an 8 x 8 chess board other than 8 x 8 is the normal size of a chess board. What is important is that the board be

square and that the number of queens to be placed be equal to the *size* of the board. We can generalize the problem by replacing the eight's with N (an arbitrary positive integral value). It follows that a 3 x 3 board would need 3 queens, a 4 x 4 board 4 queens, etc.

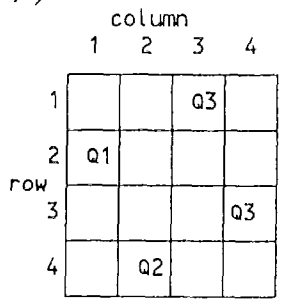
It is instructive to look at smaller board configurations as they are easier to manipulate and their properties may be applicable to larger boards. If we had a 4 x 4 chess board labeled as in the figure below, a queen placed on square 2 of the first column could legally move horizontally to square 6, 10, or 14; vertically to square 1, 3, or 4; and diagonally to square 5, 7, or 12.

1	5	9	13
Q	6	10	14
3	7	11	15
4	8	12	16

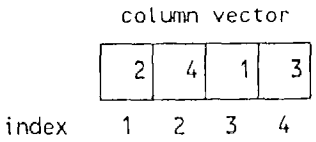
If we wanted to place a second queen on this 4 x 4 board, six squares would still be available: 8, 9, 11, 13, 15 and 16. All other squares either contain the initial queen or are accessible by it. By placing the second queen at square 8 (the only free square in the second column) the remaining available squares are reduced to: 9, 13, and 15. To place a third queen in the third column we must put it at square 9 as this is the only free square in this column. In turn this forces the fourth queen to be placed at location 15 in the fourth column. By doing so we produce the solution below.

		Q	
Q			
			Q
	Q		

To meet the condition that no queen can be in check by any other queen, we have ended up placing one queen in each row and column of the board. This one-to-one relationship is sometimes quaintly referred to as the pigeon hole principle (only one pigeon to each hole). If we change our numbering to reflect row and column location, we see that the first queen (moving from left to right) is at row 2, column 1 (2,1). The second queen is at row 4, column 2 (4,2), the third queen at row 1, column 3 (1,3) and the fourth queen at row 3, column 4 (3,4).



Considering that we can only have one queen in each column we can reduce the row / column notation to a single 4 element column vector that contains only the row locations. With this notation the solution above could be thought of as:



We can further simplify this to the sequence: 2, 4, 1, 3. Here the column locations are implied from the index location of the row value in the sequence. With pencil and paper and a small amount of trial and error we should be able to find the second valid solution for a 4 x 4 board. The two solutions are:

1. 2, 4, 1, 3 2. 3, 1, 4, 2

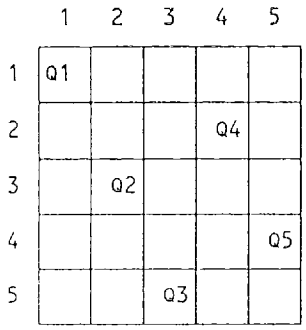
Looking at the two solutions we note that second solution is the inverse of the first. The basis for this lies in the reflective nature of the chess board. If we were to flip the chess board over (assume the queens are attached) and *look through* the board, we would see that the first solution becomes the second solution. Therefore, for any solution a second mirror image solution can be produced by flipping over the board. This property is applicable to boards of all sizes.

If we investigate a 5 x 5 board, other interesting properties are apparent. The 10 solutions for a 5 x 5 board are:

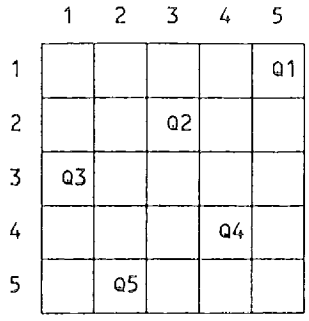
<- inverted ->

1. 1, 3, 5, 2, 4	6. 4, 2, 5, 3, 1
2. 1, 4, 2, 5, 3	7. 3, 5, 2, 4, 1
3. 2, 4, 1, 3, 5	8. 5, 3, 1, 4, 2
4. 2, 5, 3, 1, 4	9. 4, 1, 3, 5, 2
5. 3, 1, 4, 2, 5	10. 5, 2, 4, 1, 3

In pictorial format the first solution: 1, 3, 5, 2, 4 is:



If we rotate the board 90 degrees clockwise (keeping the queens attached), the queens are now at: 3, 5, 2, 4, 1 (solution number 7 in the list above).



Rotating the board an additional 90 degrees (180 degrees from the start) moves the queens to: 2, 4, 1, 3, 5 (solution number 3). A final 90 degree rotation produces: 5, 2, 4, 1, 3 (solution number 10). Therefore, starting with the first solution and rotating the board three times we generated a total of eight solutions (four solutions plus their inverses).

If we look at the sequences that arise from the first sequence via the rotations, an interesting pattern can be observed. Using the first sequence as a base (1, 3, 5, 2, 4), the next sequence produced by rotation is: 3, 5, 2, 4, 1. This sequence is the same as the first sequence except the initial value of 1 has been rotated right to the last position. If we continue to remove the first value and place it at the rear of the sequence, we in turn generate the remaining sequences produced by the 90 degree rotations (albeit in a slightly different order). Of even greater interest is the fact that if we continue this process until we return to the same starting value (i.e., 1), we generate a new sequence not produced by the 90 degree rotations: 4, 1, 3, 5, 2 (solution number 9). When this last solution is paired with its inverse it completes the list of the ten valid solutions for the 5 x 5 board.

How nice and neat things would be if the solutions for boards of every size exhibited such characteristics. Unfortunately this is not so. If we attempt to apply the 90 degree rotation technique to our initial 4 x 4 board solution, it does not work. No matter which way we rotate the board, 90, 180 or 270 degrees, we still obtain the same sequence. If we implement the rotation using our second technique by moving the first value of the sequence to the last position (i.e., 2, 4, 1, 3 becomes 4, 1, 3, 2), we end up placing location values next to one another that only differ by 1 (e.g., the 3, 2 sequence). Owing to the nature of the board, these values indicate locations of queens that will be on the diagonal from one another. It is apparent that not all boards and sequences exhibit the symmetry shown by the 5 x 5 board. While this lack of consistency is somewhat distressing, during our discussion so far we have found:

1. There can be only one queen in each row. More precisely a queen at row r threatens any other queen at row r' if r = r'. By incorporating this constraint into our solution for a board of size N, we can produce N^N possible solutions.

2. There can be only one queen in each column. Therefore, a queen at column c threatens any other queen at column c' if c = c'. This constraint, when combined with the first, will further reduce the

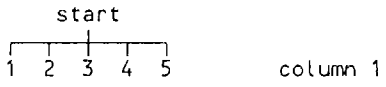
number of possible solutions to N! (i.e., all the permutations of the values 1 - N).

3. Queens cannot reside on the same diagonal. Using the previous r, c notation, queens threaten if r + c = r' + c' (on same major diagonal) or if r - c = r' - c' (same minor diagonal). The addition of this constraint to the first two constraints will reduce further the number of solutions.

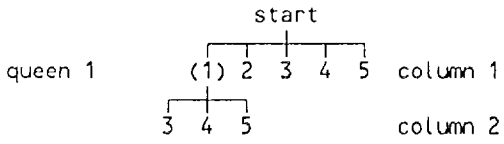
4. All valid solutions have a *mirror image* inverse. Thus, the number of solutions for any board should be divisible by two.

5. Some, but not all, solution sequences exhibit additional symmetric characteristics.

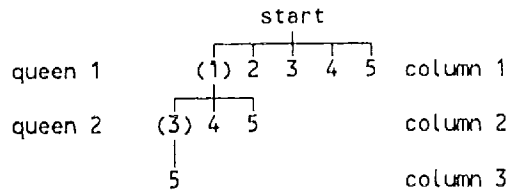
Sometimes if we change our viewpoint, we find new aspects to a problem. We can view our chess problem as a tree. If we start with the 5 x 5 board and keep track of unbounded nodes (unbounded nodes contain locations that have not been removed due to boundary constraints), our tree would be:



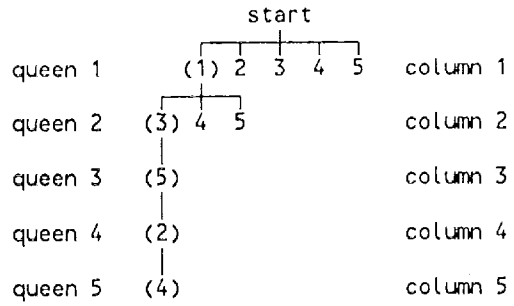
This tree shows that initially (in column one) we can place a queen at any of the five locations (rows). If we choose to place the first queen at 1 (row 1, column 1) as in our first 5 x 5 solution given previously, our tree becomes:



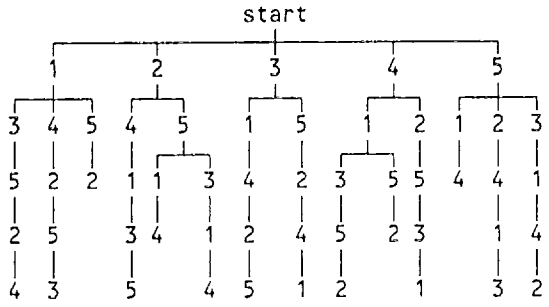
Placing the first queen in row one causes locations 1 and 2 to be out of bounds for the second queen (location 1 as queens cannot be on the same row, and location 2 as queens cannot be on the same diagonal). Choosing location 3 (row 3, column 2) produces the tree:



At this point our boundary conditions are such that each ensuing node can have only one leaf. Completing our sequence, the tree becomes:



The full unbounded tree for all the solutions to the 5 x 5 board is shown below:



Note that only the 10 paths to the lowest level (for a 5 x 5 board this would be level 5) show successful solutions. Other paths (e.g., 1,5,2), while initially containing unbounded nodes, lead to dead ends.

If we disregard the start and count the nodes (where each node represents a board location), we find a total of 53 nodes in the tree. This tree is considerably smaller (will contain fewer nodes) than the tree that would be generated if no boundary conditions were applied. Staying with a 5 x 5 board with no boundary conditions, there are 25 possible locations for placement of the first queen. The second queen also can be placed at any one of the 25 board locations (remember that with no boundary conditions queens can, in theory, be placed on top of one another). It follows in turn that the third, fourth and fifth queens each have 25

possible locations where they can be placed. This tree, were we to attempt to diagram it, would be impressive indeed!

When viewing the chess problem as a tree we found:

1. Only the paths that lead to the lowest level (level N) of the tree are successful solutions.

2. The application of boundary conditions (constraints) greatly reduces the size of the tree we generate. Boundary conditions produce the greatest reductions when applied early on in the construction of the tree.

4. A Solution

From what we have seen it would appear best to view the chess problem at the abstract level as a tree. Nodes (board locations) would be added to the tree based on the boundary conditions noted earlier. By traversing the tree using a depth-first search, valid solutions for a board size of N x N (if present) would be the path sequences to level N of the tree.

When we map these abstractions and our previous observations into code, a very good, compact solution can be generated. A complete solution to the problem, written in C and described briefly below, can be found in **Figure 1**. In writing the solution I chose to break the code into four functions which are briefly described as:

main( ) - Obtains initial settings (such as board size, etc.) and then calls function **find_sol( )**.

find_sol( ) - Finds all valid solutions for the given board. Each new row, column location for a queen is tested for boundary conditions by calling the function **no_conflict( )**. When a solution sequence is found, the function **print_it( )** is called to display the solution in the selected manner.

no_conflict( ) - Checks the row value passed in to determine if the specified location is in conflict with any previously placed queen. The function returns a TRUE if no conflict arises otherwise it returns a FALSE.

print_it( ) - Displays the solution found in either a numeric or diagrammatic (chess board) format.

```
/*
    Queen Solution Program
    John S. Gray      1993
*/

typedef enum {FALSE, TRUE} BOOLEAN;
#define MAX 8                   /* Largest board size          */
                                /*         GLOBALS             */
int a[MAX],                     /* vector holding solution     */
    size = 0,                   /* board size selected by user */
    type = 0,                   /* display type                */
    numb = 0;                   /* total number of solutions   */

void
  main( ) {

  void find_sol( int, int );

  printf("\nQueen demonstration\n");
  do {                                  /* Obtain board size   */
     printf("\n\n\nEnter the size of the board 1-%ld  ", MAX);
     scanf("%d", &size );
  } while ( size < 0 || size > MAX );
  do {                                  /* Obtain output format */
     printf("Enter the output display type 1 = diagrammatic \
             2 = numeric ");
     scanf("%d", &type );
  } while ( type < 0 || type > 2 );

  printf("\n\nFor a board of size %d solutions are: \n",size);
  find_sol( 0 , 0 );                    /* Initial call        */
  if ( ! numb ) printf("\n\nZERO !");
  printf("\n\n\n");
}


/*
  Function find_sol: Passed row and column position.
                     Generates locations for "size" number of
                     queens.  Uses recursion to backtrack.
*/

void
find_sol( int row, int col ){

  void print_it( );
  BOOLEAN no_conflict( int, int );

  if (no_conflict(row, col)){/* Check for conflicts, if none*/
    a[col]=row;              /* save the location in vector.*/
    if ( col == size-1 )
      print_it( );           /* If full solution display.   */
    else                     /* Otherwise reset the row and */
      find_sol(0, col+1);    /* check the next column.      */
    }
  if (row < size-1)          /* If more rows are available, */
    find_sol(row+1, col);    /* try the next row same column*/
}


/*
   Function no_conflict: Passed row and column position. Returns
                         TRUE if no conflict with previous
                         locations else returns FALSE
*/

BOOLEAN
no_conflict( int row, int col ){

 register i;
 int d;                  /* Temporary diagonal offset value.*/
 BOOLEAN ok = TRUE;      /* Assume no conflict at the start.*/

/*
   Step backward and check for conflicts with
   previous selections.
*/

for ( i = col-1; i >= 0 && ok ; --i ) {
  d = col - i;
  if ( a[i]   == row || /* Check for conflict in: same row,*/
       a[i]-d == row || /* same major diagonal,            */
       a[i]+d == row    /* same minor diagonal.            */
     ) ok = FALSE;
  }
  return( ok );
}
```

```
/*
   Function print_it : Displays output in user selected format.
*/

void
print_it( ){
 register r, c;
 static char line[]="+---+---+---+---+---+---+---+---+";
 line[size*4+1] = '\0';              /* cut string to size */

 printf("\n%03d : ", ++numb );

 switch ( type ) {
   case 1:
       printf("\n\n%s\n", line );
       for ( r = 0; r < size ; ++r ) {
         for ( c = 0; c < size ; ++c )
           printf("| %c ", a[r] == c ? 'Q' : (r+c) % 2 ? '#':' ');
         printf("|\n%s\n", line );
       }
       break;
   case 2  :
   default :
       for ( c = 0; c < size ; ++c )
         printf("%3d", a[c]+1 );
   }
}
```

Figure 1
Listing of Queen Program

While most of the program is self explanatory, the function **find_sol( )** does bear some further interpretation. **Find_sol( )** traverses the tree recursively in a depth-first manner. Each queen location generated is tested against a set of constraints found in the function **no_conflict( )**. If the queen can be added at the specified location this information is stored in a global vector called **a[ ]**. If an entire solution sequence has been found, it is displayed. Otherwise the function resets the value in row to its initial value and increments the column location and continues by calling itself. If the queen cannot be added, the next row in the same column is checked. If all row locations in a given column are invalid, the function returns (backtracks) to the previous column location, increments the row value for this location and begins again. The recursive nature of the function provides the *memory* for previous row values. The function will construct, prune, traverse and display the tree. For comparison a second non-recursive version of the **find_sol( )** function is shown in **Figure 2**. The flow of logic for this function is very similar to the first **find_sol( )** function. However, in this version of the function, backtracking is facilitated by using the vector **a[ ]** as a stack. Previous queen locations that have been added (pushed on) to the stack are popped when needed, using the while loop at the foot of the function.

```
/*
   A non-recursive find_sol( ) function.  A few more lines of
   code but less system intensive.
*/


void
find_sol( int row, int col ){
  void print_it( );
  BOOLEAN no_conflict( int, int );
  do {
   if (no_conflict(row,col)){ /*Check for conflicts, if none*/
     a[col] = row;            /*save the location in vector.*/
     if ( col==size-1 )       /*If full solution display.   */
       print_it( );
     else {                   /*Otherwise reset the row and */
       row = 0;  ++col;       /*check the next column.      */
       continue;
     }
   }
   if ( row < size-1 )        /*If more rows are available  */
     ++row;                   /*try next row, same column.  */
   else {                     /*Otherwise backtrack until a */
     while (col && a[col-1]+1 > size-1)
       --col;                 /*column is found with a row  */
     row = a[--col]+1;        /*value that can be increment-*/
   }                          /*ed to the next row.         */
  } while ( row < size );     /*Loop until beyond last row  */
}                             /*in the first column.        */
```

Figure 2
Non-recursive find_sol( ) Function

A sample run of the program is shown in **Figure 3**. I should note that the program solution does not take into account the symmetries discussed earlier. Those who wish to pursue the problem further should review the paper by Sosic and Gu [2]. The authors present a unique, polynomial time, non exhaustive algorithm for the generation of solutions for large queen problem sets.

```
Queen demonstration


Enter the size of the board 1-8   6

Enter the output display type 1 = diagrammatic, 2 = numeric 2

For a board of size 6 the solutions are:

001 :   2  4  6  1  3  5
002 :   3  6  2  5  1  4
003 :   4  1  5  2  6  3
004 :   5  3  1  6  4  2
```

Figure 3
Demonstration of Queen Program

5. Conclusion

When generating solutions to computer problems, as in life, Walter C. Hagen's advice *"Don't hurry, don't worry. You're only here for a short visit. So be sure to stop and smell the flowers."* is apropos. The extra time taken to analyze a problem can lead to insights into the problem which in turn will result in cleaner more efficient solutions.

****EIGHT ENOUGH REFERENCES On Page 51****

compiler that they really know what they are doing. Simply put, the programmer's motto should be 'Plan Your Program and Program Your Plan.'

Hints and Conclusion

As with all classes, it is important to know your audience. A considerable number of "grad" students take beginning C classes. These students already have a degree(s) and careers but want to know C. The instructor needs to be careful not to focus on the interests of these higher-level students. This approach quickly alienates and loses the undergraduates the course was designed to serve. If willing, the "grads" can serve as mentors for some of the traditional students.

Picking C software is somewhat more difficult as both Microsoft and Borland move to blend C and C++ product lines together. Using a C++ product in a C class would allow the use of new comment formats (//... instead of /*...*/) and I/O functions (cin/cout instead of scanf/printf) albeit at the expense of backwards compatibility. Perhaps more useful, will be the possibility of having a two-class sequence, ANSI C followed by C++ for object-oriented programming, where the students would use the same compiler and editor for both.

Finally, the C instructor should explain advantages and disadvantages of various approaches, showing more than one right way to accomplish a task. Code from the book can be modified on-the-run if a computer projection system is in use. Another way to show multiple approaches to problems where each student fully understands the specifications is to post student programming assignments (two source code listings are turned in -- one is marked during grading and returned to the student, the other is available for taping to the board). Such "good examples" may be demonstrated by the author in class if a computer projection system is available.

References

The American Heritage Dictionary. (1985). Boston: Houghton Mifflin Company.

Baldwin, W. Debugging In C -- An Overview. C Users Journal, 9(10). (Computer Select, October 1992, #74755)

Pournelle, J. The BYTE Summit: Obstacles to Overcome. BYTE, September 1990, 281.

Kernighan, B. W., & Ritchie, D. M. (1978). The C Programming Language. Englewood Cliffs, NJ: Prentice-Hall, Inc., 76.

Norton, P. PC Magazine, February 10, 1987, 75-76.

Plauger, P. Bugs. C Users Journal, 10(9). (Computer Select, October 1992, #2055)

Figure drawn by MET student Paul Pepka.

EIGHT ENOUGH REFERENCES From Page 44

6. References

[1] R. L. Kruse, Data Structures & Program Design, Prentice Hall, Englewood Cliffs, New Jersey, 1987, 586 pages.

[2] R. Sosic and J. Gu, "A Polynomial Time Algorithm for the N-Queens Problem", SIGART Bulletin, Vol. 1, No. 3, October 1990, 7-11.

[3] N. Wirth, Algorithms + Data Structures = Programs, Prentice Hall, Englewood Cliffs, New Jersey, 1976, 366 pages.